Lexical Analysis: What is it?

Adapted from material by:
Prof. Alex Aiken and Prof. George Necula (UCB)
Prof. Saman Amarasinghe (MIT)

Outline
• Informal sketch of lexical analysis
  ➢ Tokens vs. Lexemes vs. Attributes
• Issues in lexical analysis
  ➢ Lookahead
  ➢ Ambiguities
• Specifying lexers
  ➢ Regular expressions
  ➢ Examples of regular expressions

Lexical Analysis
• What do we want to do? Example:
  if (i == j)
    Z = 0;
  else
    Z = 1;
• The input is just a string of characters:
  \[ if (i == j) \ n \ t \ z = 0; \ n \ telse\ n \ t \ t z = 1; \]
• Goal: Partition input string into substrings (lexemes) to identify tokens

Source program text

Tokens
• Examples of Tokens
  ➢ Operators
  ➢ Keywords: if while for int double
  ➢ Numeric literals
  ➢ Character literals
  ➢ String literals
  ➢ Examples of non-tokens
  ➢ White space
  ➢ Comments: /*this is not a token*/
Token Type vs. Lexeme vs. Attribute

- **Token type**: A syntactic category/grouping
  - In English: noun, verb, adjective, ...
  - In a programming language: identifier, integer, keyword, ';', '[', ...
- **Lexeme**: Concrete manifestation of a token in the text.
  - In a case-insensitive language, the lexemes associated with the IF token are: if, IF, iF, and If.
- **Attribute**: “Value of interest” about a token.
  - Numerical value of an integer token.
  - Name (string) associated with an identifier token.

Lexical Analyzer (Lexer/Scanner/Tokenizer)

- **Designing a Lexical Analyzer**
  1. Define a finite set of tokens.
  2. Describe which strings belong to each token.
- **Implementing a Lexical Analyzer**
  - Recognize tokens from the corresponding lexemes.
  - Return the value (attribute) and the type of the token.
    - 6036  Num(6036)
    - X6035  ID(“X6035”)
  - Eliminate whitespaces, comments, ...etc that do not contribute to parsing.

Example: Language Design Decisions

- **FORTRAN rule**: Whitespace is insignificant.
  - E.g., VAR1 is the same as V AR1
  - Consider
    - DO 5 I = 1,25
    - DO 5 I = 1.25
      - The first is DO 5 I = 1 , 25
      - The second is DO$5$ I = 1.25
    “Lookahead” may be required to decide where one token ends and the next token begins.
  - Even our simple examples have lookahead issues.
    - if
    - $==$ vs. $==$  vs.
    - Earlier DO-example

- **PL/I keywords are not reserved**: IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
- Ada and VHDL require 2-lookahead because of the tick (‘’) problem.
- In Ada, array reference syntax and function call syntax are similar.
  - arr(2,3) vs. fn(1,2)
- In C++, these are different.
  - arr[2,3] vs. fn[1,2]
• C++ template syntax:
  Foo<Bar>
• C++ stream syntax:
  cin >> var;
• But there is a conflict with nested templates:
  Foo<Bar<Bazz>>
• In general, complexity stems from the interaction among the various features.

Specifying Lexical Structure

• We need
  ➢ A way to describe the lexemes of each token.
  ➢ A way to resolve ambiguities.
    ➢ Is if two variables i and f?
    ➢ Is two equal signs ==?
• Typically, lexemes associated with a token (type) form a regular language. So, use Regular Expressions to specify tokens.

Example: Phone Numbers

• Consider (937)-775-5134

Σ = digits ∪ {-,(.)}
exchange = digit 3
phone = digit 4
area = digit 3
phone_number = '(' area ')'-exchange '-' phone

Example: Email Addresses

• Consider violin@cs.wright.edu

Σ = letters ∪ {.,@}
name = letter*
address = name '@' name '.' name '.' name


Definition: Formal Languages

- Alphabet \( \Sigma = \) finite set of symbols
  \( \Sigma = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \} \)
- String \( s = \) finite sequence of symbols from the alphabet
  \( s = 6004 \)
- Empty string \( \epsilon = \) special string of length zero
- Language \( L = \) set of strings over an alphabet
  \( L = \{ 6001, 6002, 6003, 6004, 6035, 6891 \ldots \} \)

Regular Expressions over \( \Sigma \)

- Basis: \( \phi, \epsilon, \) and \( \forall a \in \Sigma: \alpha \) are regular expressions over \( \Sigma \).
- Inductive Step: Let \( r \) and \( s \) be regular expressions over \( \Sigma \). Then so are:
  \( r | s, rs, \) and \( r^* \).
- Closure: Nothing else is a regular expression, unless obtained using the above steps.

Integer Power of a Language

- \( X = \{ 0, 1 \} \)
- \( X^0 = \{ \epsilon \} \)
- \( X^1 = \{ 0, 1 \} \)
- \( X^r = X^r \cup X^r \cup \ldots \cup X^r \cup \ldots \)
- \( X^* = \{ \text{bit strings of length } n \} \)

Kleene Star

- Note that \( \epsilon \in X \Rightarrow \epsilon \in X^* \)

Kleene Plus

- Syntax vs Semantics

- Regular Expressions (numeral)
  \( \phi \), \( \epsilon \), \( \phi \), \( 0 \), \( \epsilon 0 \), \( 0 \epsilon \)
- Regular sets/language (number)
  \( \{ \phi \} \), \( \{ \epsilon \} \), \( \{ 0 \} \)
Semantics of Regular Expressions

• \( L : \text{reg-expr} \rightarrow \text{set of strings} \)
  - \( L((a | \varepsilon) \cdot b) = \{ \text{“ab”}, \text{“b”} \} \)

• Suppose \( r \) and \( s \) are regular expressions denoting languages \( L(r) \) and \( L(s) \)
  - \( L(r | s) = L(r) \cup L(s) \) (union)
  - \( L(r \cdot s) = L(r) \cdot L(s) \) (concatenation)
  - \( L(r^*) = \bigcup_{n=0}^{\infty} L^*(r) \) (Kleene Closure)
  - \( L(\varepsilon) = \{ \varepsilon \} \)
  - \( L(\phi) = \{ \} \)
  - \( L(a) = \{ \text{“a”} \} \) for all \( a \) in \( \Sigma \)

• Few additional ones used in FLEX
  - “one or more occurrences of” \( r^+ = r \cdot r^* \)
  - “zero or one occurrence of” \( r? = r | \varepsilon \)
  - “negated character class” \( [\^A-Z] \)
    - Any character EXCEPT an uppercase letter.
  - “an \( r \) but only if it is followed by an \( s \)” \( r/s \)

• These additional operators correspond to closure operations on regular languages.